

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

Protecting Digital Goods Using Oblivious Checking

Inventor(s):

Mariusz H. Jakubowski
Ramarathnam Venkatesan

ATTORNEY'S DOCKET NO. MS1-516US

1 **REFERENCE TO RELATED APPLICATIONS**

2 This is a continuation-in-part of Application No. 09/536,033, filed March
3 27, 2000, entitled "System and Method for Protecting Digital Goods Using
4 Random and Automatic Code Obfuscation".

5
6 **TECHNICAL FIELD**

7 This invention relates to systems and methods for protecting digital goods,
8 such as software.

9
10 **BACKGROUND**

11 Digital goods (e.g., software products, data, content, etc.) are often
12 distributed to consumers via fixed computer readable media, such as a compact
13 disc (CD-ROM), digital versatile disc (DVD), soft magnetic diskette, or hard
14 magnetic disk (e.g., a preloaded hard drive). More recently, more and more
15 content is being delivered in digital form online over private and public networks,
16 such as Intranets and the Internet. Online delivery improves timeliness and
17 convenience for the user, as well as reduces delivery costs for a publisher or
18 developers. Unfortunately, these worthwhile attributes are often outweighed in the
19 minds of the publishers/developers by a corresponding disadvantage that online
20 information delivery makes it relatively easy to obtain pristine digital content and
21 to pirate the content at the expense and harm of the publisher/developer.

22 One concern of the publisher/developer is the ability to check digital
23 content, after distribution, for alteration. Such checking, is often referred to as
24 SRI (Software Resistance to Interference). The desire to check for such alterations
25

1 can vary (e.g., to ensure that the content continues to operate as intended by the
2 publisher/developer, to protect against improper copying, etc.).

3 The unusual property of content is that the publisher/developer (or reseller)
4 gives or sells the *content* to a client, but continues to restrict *rights* to use the
5 content even after the content is under the sole physical control of the client. For
6 instance, a software developer typically sells a limited license in a software
7 product that permits a user to load and run the software product on one or more
8 machines (depending upon the license terms), as well as make a back up copy.
9 The user is typically not permitted to make unlimited copies or redistribute the
10 software to others.

11 These scenarios reveal a peculiar arrangement. The user that possesses the
12 digital bits often does not have full rights to their use; instead, the provider retains
13 at least some of the rights. In a very real sense, the legitimate user of a computer
14 can be an adversary of the data or content provider.

15 One of the uses for SRI is to provide “digital rights management” (or
16 “DRM”) protection to prevent unauthorized distribution of, copying and/or illegal
17 operation of, or access to the digital goods. An ideal digital goods distribution
18 system would substantially prevent unauthorized distribution/use of the digital
19 goods. Digital rights management is fast becoming a central requirement if online
20 commerce is to continue its rapid growth. Content providers and the computer
21 industry must quickly address technologies and protocols for ensuring that digital
22 goods are properly handled in accordance with the rights granted by the
23 developer/publisher. If measures are not taken, traditional content providers may
24 be put out of business by widespread theft or, more likely, will refuse altogether to
25 deliver content online.

1 Various DRM techniques have been developed and employed in an attempt
2 to thwart potential pirates from illegally copying or otherwise distributing the
3 digital goods to others. For example, one DRM technique includes requiring the
4 consumer to insert the original CD-ROM or DVD for verification prior to enabling
5 the operation of a related copy of the digital good. Unfortunately, this DRM
6 technique typically places an unwelcome burden on the honest consumer,
7 especially those concerned with speed and productivity. Moreover, such
8 techniques are impracticable for digital goods that are site licensed, such as
9 software products that are licensed for use by several computers, and/or for digital
10 goods that are downloaded directly to a computer. Additionally, it is not overly
11 difficult for unscrupulous individuals/organizations to produce working pirated
12 copies of the CD-ROM.

13 Another DRM technique includes requiring or otherwise encouraging the
14 consumer to register the digital good with the provider, for example, either through
15 the mail or online via the Internet or a direct connection. Thus, the digital good
16 may require the consumer to enter a registration code before allowing the digital
17 good to be fully operational or the digital content to be fully accessed.
18 Unfortunately, such DRM techniques are not always effective since unscrupulous
19 individuals/organizations need only break through or otherwise undermine the
20 DRM protections in a single copy of the digital good. Once broken, copies of the
21 digital good can be illegally distributed, hence such DRM techniques are
22 considered to be Break-Once, Run-Everywhere (BORE) susceptible. Various
23 different techniques can be used to defeat BORE, such as per-user software
24 individualization, watermarks, etc. However, a malicious user may still be able to
25 identify and remove from the digital good these various protections.

1 Accordingly, there remains a need for a technique that addresses the
2 concerns of the publisher/developer, allowing alteration of the digital content to be
3 identified to assist in protecting the content from many of the known and common
4 attacks, but does not impose unnecessary and burdensome requirements on
5 legitimate users.

6 7 SUMMARY

8 Oblivious checking of digital goods is described herein.

9 According to one aspect, a plurality of key instructions within a function of
10 a digital good are identified. Each key instruction is an instruction that possibly
11 modifies a register or a flag. An extra instruction is then inserted into the function
12 for each of the key instructions. The extra instructions each correspond to one of
13 the key instructions and modify a register in a deterministic fashion based on the
14 corresponding key instruction.

15 A set of inputs to the function are then identified that result in different
16 valid computation paths in the function being taken. A checksum for the function
17 is then generated by using a mapping function which maps the contents of the
18 register to the set of inputs.

19 According to another aspect, the extra instructions are added to the function
20 in appropriate locations such that the extra instruction corresponding to a key
21 instruction is always executed if the corresponding key instruction is executed, and
22 is executed after the corresponding key instruction is executed.

BRIEF DESCRIPTION OF THE DRAWINGS

The same numbers are used throughout the drawings to reference like elements and features.

Fig. 1 is a block diagram of a DRM distribution architecture that protects digital goods by automatically and randomly obfuscating portions of the goods using various tools.

Fig. 2 is a block diagram of a system for producing a protected digital good from an original good.

Fig. 3 is a flow diagram of a protection process implemented by the system of Fig. 2.

Fig. 4 is a diagrammatical illustration of a digital good after being coded using the process of Fig. 3.

Fig. 5 is a diagrammatical illustration of a protected digital good that is shipped to a client, and shows an evaluation flow through the digital good that the client uses to evaluate the authenticity of the good.

Fig. 6 is a flow diagram of an oblivious checking process that may be employed by the system of Fig. 2.

Fig. 7 is a diagrammatic illustration of a digital good that is modified to support code integrity verification.

Fig. 8 is a diagrammatic illustration of a digital good that is modified to support cyclic code integrity verification.

DETAILED DESCRIPTION

A digital rights management (DRM) distribution architecture produces and distributes digital goods in a fashion that renders the digital goods resistant to

1 many known forms of attacks. The DRM distribution architecture protects digital
2 goods by automatically and randomly manipulating portions of the code using
3 multiple protection techniques. Essentially any type of digital good may be
4 protected using this architecture, including such digital goods as software, audio,
5 video, and other content. For discussion purposes, many of the examples are
6 described in the context of software goods, although most of the techniques
7 described herein are effective for non-software digital goods, such as audio data,
8 video data, and other forms of multimedia data.

DRM Distribution Architecture

9
10
11 Fig. 1 shows a DRM distribution architecture 100 in which digital goods
12 (e.g., software, video, audio, etc.) are transformed into protected digital goods and
13 distributed in their protected form. The architecture 100 has a system 102 that
14 develops or otherwise produces the protected good and distributes the protected
15 good to a client 104 via some form of distribution channel 106. The protected
16 digital goods may be distributed in many different ways. For instance, the
17 protected digital goods may be stored on a computer-readable medium 108 (e.g.,
18 CD-ROM, DVD, floppy disk, etc.) and physically distributed in some manner,
19 such as conventional vendor channels or mail. The protected goods may
20 alternatively be downloaded over a network (e.g., the Internet) as streaming
21 content or files 110.

22 The developer/producer system 102 has a memory 120 to store an original
23 digital good 122, as well as the protected digital good 124 created from the
24 original digital good. The system 102 also has a production server 130 that
25 transforms the original digital good 122 into the protected digital good 124 that is

1 suitable for distribution. The production server 130 has a processing system 132
2 and implements an obfuscator 134 equipped with a set of multiple protection tools
3 136(1)-136(N). Generally speaking, the obfuscator 134 automatically parses the
4 original digital good 122 and applies selected protection tools 136(1)-136(N) to
5 various portions of the parsed good in a random manner to produce the protected
6 digital good 124. Applying a mixture of protection techniques in random fashion
7 makes it extremely difficult for pirates to create illicit copies that go undetected as
8 legitimate copies.

9 The original digital good 122 represents the software product or data as
10 originally produced, without any protection or code modifications. The protected
11 digital good 124 is a unique version of the software product or data after the
12 various protection schemes have been applied. The protected digital good 124 is
13 functionally equivalent to and derived from the original data good 122, but is
14 modified to prevent potential pirates from illegally copying or otherwise
15 distributing the digital goods to others. In addition, some modifications enable the
16 client to determine whether the product has been tampered with.

17 The developer/producer system 102 is illustrated as a single entity, with
18 memory and processing capabilities, for ease of discussion. In practice, however,
19 the system 102 may be configured as one or more computers that jointly or
20 independently perform the tasks of transforming the original digital good into the
21 protected digital good.

22 The client 104 has a secure processor 140, memory 142 (e.g., RAM, ROM,
23 Flash, hard disk, CD-ROM, etc.), one or more input devices 144 (e.g., keyboard,
24 joystick, voice recognition, etc.), and one or more output devices 146 (e.g.,
25 monitor, speakers, etc.). The client may be implemented as a general purpose

1 computing unit (e.g., desktop PC, laptop, etc.) or as other devices, such as set-top
2 boxes, audio/video appliances, game consoles, and the like.

3 The client 104 runs an operating system 150, which is stored in memory
4 142 and executed on the secure processor 140. Operating system 150 represents
5 any of a wide variety of operating systems, such as a multi-tasking, open platform
6 system (e.g., a "Windows"-brand operating system from Microsoft Corporation).
7 The operating system 150 includes an evaluator 152 that evaluates the protected
8 digital goods prior to their utilization to determine whether the protected digital
9 goods have been tampered with or modified in any manner. In particular, the
10 evaluator 152 is configured to analyze the various portions according to the
11 different protection schemes originally used to encode the good to evaluate the
12 authenticity of the digital good.

13 Some protection schemes involve executing instructions, analyzing data,
14 and performing other tasks in the most secure areas of the operating system 150
15 and secure processor 140. Accordingly, the evaluator 152 includes code portions
16 that may be executed in these most secure areas of the operating system and secure
17 processor. Although the evaluator 152 is illustrated as being integrated into the
18 operating system 150, it may be implemented separately from the operating
19 system.

20 In the event that the client detects some tamper activity, the secure
21 processor 140 acting alone, or together with the operating system 150, may decline
22 to execute the suspect digital code. For instance, the client may determine that the
23 software product is an illicit copy because the evaluations performed by the
24 evaluator 152 are not successful. In this case, the evaluator 152 informs the secure
25

1 processor 140 and/or the operating system 150 of the suspect code and the secure
2 processor 140 may decline to run that software product.

3 It is further noted that the operating system 150 may itself be the protected
4 digital good. That is, the operating system 150 may be modified with various
5 protection schemes to produce a product that is difficult to copy and redistribute,
6 or at least makes it easy to detect such copying. In this case, the secure processor
7 140 may be configured to detect an improper version of the operating system
8 during the boot process (or at other times) and prevent the operating system from
9 fully or partially executing and obtaining control of system resources.

10 For protected digital goods delivered over a network, the client 104
11 implements a tamper-resistant software (not shown or implemented as part of the
12 operating system 150) to connect to the server 102 using an SSL (secure sockets
13 layer) or other secure and authenticated connection to purchase, store, and utilize
14 the digital good. The digital good may be encrypted using well-known algorithms
15 (e.g., RSA) and compressed using well-known compression techniques (e.g., ZIP,
16 RLE, AVI, MPEG, ASF, WMA, MP3).

17 18 **Obfuscating System**

19 Fig. 2 shows the obfuscator 134 implemented by the production server 130
20 in more detail. The obfuscator 134 is configured to transform an original digital
21 good 122 into a protected digital good 124. The obfuscating process is usually
22 applied just before the digital good is released to manufacture or prior to being
23 downloaded over a network. The process is intended to produce a digital good
24 that is protected from various forms of attacks and illicit copying activities. The
25

1 obfuscator 134 may be implemented in software (or firmware), or a combination
2 of hardware and software/firmware.

3 The obfuscator 134 has an analyzer 200 that analyzes the original digital
4 good 122 and parses it into multiple segments. The analyzer 200 attempts to
5 intelligently segment the digital good along natural boundaries inherent in the
6 product. For instance, for a software product, the analyzer 200 may parse the code
7 according to logical groupings of instructions, such as routines, or sub-routines, or
8 instruction sets. Digital goods such as audio or video products may be parsed
9 according to natural breaks in the data (e.g., between songs or scenes), or at
10 statistically computed or periodic junctures in the data.

11 In one specific implementation for analyzing software code, the analyzer
12 200 may be configured as a software flow analysis tool that converts the software
13 program into a corresponding flow graph. The flow graph is partitioned into many
14 clusters of nodes. The segments may then take the form of sets of one or more
15 nodes in the flow graph. For more information on this technique, the reader is
16 directed to co-pending U.S. Patent Application Serial Number 09/525,694, entitled
17 "A Technique for Producing, Through Watermarking, Highly Tamper-Resistant
18 Executable Code and Resulting "Watermarked" Code So Formed", which was
19 filed March 14, 2000, in the names of Ramarathnam Venkatesan and Vijay
20 Vazirani. This Application is assigned to Microsoft Corporation and is hereby
21 incorporated by reference.

22 The segments may overlap one another. For instance, one segment may
23 contain a set of instructions in a software program and another segment may
24 contain a subset of the instructions, or contain some but not all of the instructions.
25

1 The obfuscator 134 also has a target segment selector 202 that randomly
2 applies various forms of protection to the segmented digital good. In the
3 illustrated implementation, the target selector 202 implements a pseudo random
4 generator (PRG) 204 that provides randomness in selecting various segments of
5 the digital good to protect. The target segment selector 202 works together with a
6 tool selector 206, which selects various tools 136 to augment the selected
7 segments for protection purposes. In one implementation, the tool selector 206
8 may also implement a pseudo random generator (PRG) 208 that provides
9 randomness in choosing the tools 136.

10 The tools 136 represent different schemes for protecting digital products.
11 Some of the tools 136 are conventional, while others are not. These distinctions
12 will be noted and emphasized throughout the continuing discussion. Fig. 2 shows
13 sixteen different tools or schemes that create a version of a digital good that is
14 difficult to copy and redistribute without detection and that is resistant to many of
15 the known pirate attacks, such as BORE (break once, run everywhere) attacks and
16 disassembly attacks.

17 The illustrated tools include oblivious checking 136(1), code integrity
18 verification 136(2), acyclic and cyclic code integrity verification 136(3), secret
19 key scattering 136(4), obfuscated function execution 136(5), code as an S-box
20 136(6), encryption/decryption 136(7), probabilistic checking 136(8), Boolean
21 check obfuscation 136(9), in-lining 136(10), reseeding of PRG with time varying
22 inputs 136(11), anti-disassembly methods 136(12), shadowing of relocatable
23 addresses 136(13), varying execution paths between runs 136(14), anti-debugging
24 methods 136(15), and time/space separation between tamper detection and
25 response 136(16). The tools 136(1)-136(16) are examples of possible protection

1 techniques that may be implemented by the obfuscator 134. It is noted that more
2 or less than the tools may be implemented, as well as other tools not mentioned or
3 illustrated in Fig. 2. The exemplary tools 136(1)-136(16) are described below in
4 more detail beneath the heading "Exemplary Protection Tools".

5 The target segment selector 202 and the tool selector 206 work together to
6 apply various protection tools 136 to the original digital good 122 to produce the
7 protected digital good 124. For segments of the digital good selected by the target
8 segment selector 202 (randomly or otherwise), the tool selector 206 chooses
9 various protection tools 136(1)-136(16) to augment the segments. In this manner,
10 the obfuscator automatically applies a mixture of protection techniques in a
11 random manner that makes it extremely difficult for pirates to create usable
12 versions that would not be detectable as illicit copies.

13 The obfuscator 134 also includes a segment reassembler 210 that
14 reassembles the digital good from the protected and non-protected segments. The
15 reassembler 210 outputs the protected digital good 124 that is ready for mass
16 production and/or distribution.

17 The obfuscator 134 may further be configured with a quantitative unit 212
18 that enables a producer/developer to define how much protection should be
19 applied to the digital good. For instance, the producer/developer might request
20 that any protection not increase the runtime of the product. The
21 producer/developer may also elect to set the number of checkpoints (e.g., 500 or
22 1000) added to the digital good as a result of the protection, or define a maximum
23 number of lines/bytes of code that are added for protection purposes. The
24 quantitative unit 212 may include a user interface (not shown) that allows the user
25 to enter parameters defining a quantitative amount of protection.

1 The quantitative unit 212 provides control information to the analyzer 200,
2 target segment selector 202, and tool selector 206 to ensure that these components
3 satisfy the specified quantitative requirements. Suppose, for example, the
4 producer/developer enters a predefined number of checkpoints (e.g., 500). With
5 this parameter, the analyzer 200 ensures that there are a sufficient number of
6 segments (e.g., >500), and the target segment selector 202 and tool selector 206
7 apply various tools to different segments such that the resulting number of
8 checkpoints approximates 500.

9 10 **General Operation**

11 Fig. 3 shows the obfuscation process 300 implemented by the obfuscator
12 134 at the production server 102. The obfuscation process is implemented in
13 software and will be described with additional reference to Figs. 1 and 2.

14 At block 302, the quantitative unit 212 enables the developer/producer to
15 enter quantitative requirements regarding how much protection should be applied
16 to the digital good. The developer/producer might specify, for example, how
17 many checkpoints are to be added, or how many additional lines of code, or
18 whether runtime can be increased as a result of the added protection.

19 At block 304, the analyzer/parser 200 analyzes an original digital good and
20 parses it into plural segments. The encoded parts may partially or fully overlap
21 with other encoded parts.

22 The target segment selector 202 chooses one or more segments (block 306).
23 Selection of the segment may be random with the aid of the pseudo random
24 generator 204. At block 308, the tool selector 206 selects one of the tools 136(1)-
25

1 136(16) to apply to the selected section. Selection of the tools may also be a
2 randomized process, with the assistance of the pseudo random generator 208.

3 To illustrate this dual selection process, suppose the segment selector 202
4 chooses a set of instructions in a software product. The tool selector 206 may then
5 use a tool that codes, manipulates or otherwise modifies the selected segment.
6 The code integrity verification tool 136(2), for example, places labels around the
7 one or more segments to define the target segment. The tool then computes a
8 checksum of the bytes in the target segment and hides the resultant checksum
9 elsewhere in the digital good. The hidden checksum may be used later by tools in
10 the client 104 to determine whether the defined target segment has been tampered
11 with.

12 Many of the tools 136 place checkpoints in the digital good that, when
13 executed at the client, invoke utilities that analyze the segments for possible
14 tampering. The code verification tool 136(2) is one example of a tool that inserts a
15 checkpoint (i.e., in the form of a function call) in the digital good outside of the
16 target segment. For such tools, the obfuscation process 300 includes an optional
17 block 310 in which the checkpoint is embedded in the digital good, but outside of
18 the target segment. In this manner, the checkpoints for invoking the verification
19 checks are distributed throughout the digital good. In addition, placement of the
20 checkpoints throughout the digital good may be random.

21 The process of selecting segment(s) and augmenting them using various
22 protection tools is repeated for many more segments, as indicated by block 312.
23 Once the obfuscator has finished manipulating the segments of the digital code
24 (i.e., the "no" branch from block 312), the reassembler 210 reassembles the
25 protected and non-protected segments into the protected digital good (block 314).

Fig. 4 shows a portion of the protected digital good 124 having segments i , $i+1$, $i+2$, $i+3$, $i+4$, $i+5$, and so forth. Some of the segments have been augmented using different protection schemes. For instance, segment $i+1$ is protected using tool 7. The checkpoint CP_{i+1} for this segment is located in segment $i+4$. Similarly, segment $i+3$ is protected using tool 3, and the checkpoint CP_{i+3} for this segment is located in segment $i+2$. Segment $i+4$ is protected using tool K , and the checkpoint CP_{i+4} for this segment is located in segment i .

Notice that the segments may overlap one another. In this example, segment $i+3$ and $i+4$ partially overlap, thus sharing common data or instructions. Although not illustrated, two or more segments may also completely overlap, wherein one segment is encompassed entirely within another segment. In such situations, a first protection tool is applied to one segment, and then a second protection tool is applied to another segment, which includes data and/or instructions just modified by the first protection tool.

Notice also that not all of the segments are necessarily protected. For instance, segment $i+2$ is left “unprotected” in the sense that no tool is applied to the data or instructions in that segment.

Fig. 5 shows the protected digital good 124 as shipped to the client, and illustrates control flow through the good as the client-side evaluator 152 evaluates the good 124 for any sign of tampering. The protected digital good 124 has multiple checkpoints $500(1)$, $500(2)$, ..., $500(N)$ randomly spread throughout the good. When executing the digital good 124, the evaluator 152 passes through the various checkpoints $500(1)$ - $500(N)$ to determine whether the checks are valid, thereby verifying the authenticity of the protected digital good.

1 If any checkpoint fails, the client is alerted that the digital good may not be
2 authentic. In this case, the client may refuse to execute the digital good or disable
3 portions of the good in such a manner that renders it relatively useless to the user.
4

5 Exemplary Protection Tools

6 The obfuscator 134 illustrated in Fig. 2 shows sixteen protection tools
7 136(1)-136(16) that may be used to protect the digital good in some manner. The
8 tools are typically invoked after the parser 200 has parsed the digital good into
9 multiple segments. Selected tools are applied to selected segments so that when
10 the segment good is reassembled, the resulting protected digital good is a
11 composite of variously protected segments that are extremely difficult to attack.
12 The sixteen exemplary tools are described below in greater detail.
13

14 Oblivious Checking

15 One tool for making a digital good more difficult to attack is referred to as
16 “oblivious checking”. This tool performs checksums on bytes of the digital
17 product without actually reading the bytes.

18 More specifically, the oblivious checking tool is designed so that, given a
19 function f , the tool computes a checksum $S(f)$ such that:
20

- 21 (1) If f is not changed, $S(f)$ can be verified to be correct.
- 22 (2) If f is changed to f' , $S(f') \neq S(f)$ with extremely high probability.
23

24 Fig. 6 illustrates an exemplary implementation of an oblivious checking
25 process 600 implemented by the oblivious checking tool 136(1) in the obfuscator

1 134. The first few blocks 602-606 are directed toward instrumenting the code for
2 function f. At block 602, the tool identifies instructions in the software code that
3 possibly modify registers or flags. These instructions are called "key
4 instructions". Alternatively, other instructions (or groups of instructions) could be
5 the key instructions.

6 For each key instruction, the tool inserts an extra instruction(s) that
7 modifies a register R in a deterministic fashion based on the key instruction (block
8 604). This extra instruction is placed anywhere in the code, but in one
9 implementation is placed with the requirement that it is always executed if the
10 corresponding key instruction is executed, and moreover, is always executed after
11 the key instruction. The control flow of function f is maintained as originally
12 designed, and does not change. Thus, after instrumenting the code, each valid
13 computation path of function f is expected to have instructions modifying R in
14 various ways.

15 At block 606, the tool derives an input set "I" containing inputs x to the
16 function f, which can be denoted by $I = \{x_1, x_2, x_3 \dots x_n\}$. The input set "I" may be
17 derived as a set of input patterns to function f that ensures that most or all of the
18 valid computation paths are taken. Such input patterns may be obtained, for
19 example, from profile data that provides information about typical runs of the
20 entire program. The input set "I" may be exponential in the number of branches in
21 the function, but should not be too large a number.

22 At block 608, the tool computes $S(f)$ through the use of a mapping function
23 g, which maps the contents of register R to a random element of I with uniform
24 probability. Let $f(x)$ denote the value of register R, starting with 0, after executing
25 f on input x. The function $f(x)$ may be configured to be sensitive to key features of

1 the function so that if a computation path were executed during checksum
2 computation, then any significant change in it would be reflected in $f(x)$ with high
3 probability.

4 One implementation of computing checksum $S(f)$ is as follows:

5
6 Start with $x = x_0$
7 Cks := $f(x_0)$ XOR x_0
8 For $i=1$ to K do
9 $x_i := g(f(x_{i-1}))$
10 Cks += $f(x_i)$ XOR x_i .
11 End for

12 The resulting checksum $S(f)$ is the initial value x_0 , along with the value
13 Cks, or (x_0, Cks) . Notice that the output of one iteration is used to compute the
14 input of the next iteration. This loop makes the checksum shorter, since there is
15 only one initial input instead of a set of K independent inputs (i.e., only the input
16 x_0 rather than the entire set of K inputs), although all of the K inputs need to be
17 made otherwise available to the evaluator verifying the checksum.

18 Each iteration of the loop traverses some computation path of the function
19 f . A random factor may optionally be included in determining which computation
20 path of the function f to traverse. Preferably, each computation path of function f
21 has the same probability of being examined during one iteration. For K iterations,
22 the probability of a particular path being examined is:

$$1 - (1 - 1/n)^K \approx K/n, \text{ where } n = \text{card}(I).$$

1 It should be noted that, although various randomness may be included in
2 the oblivious checking as mentioned above, such randomness should be
3 implemented in a manner that can be duplicated during verification (e.g., to allow
4 the checksum $S(f)$ to be re-calculated and verified). For example, the randomness
5 introduced by the oblivious checking tool 136(1) may be based on a random
6 number seed and pseudo-random number generator that is also known to evaluator
7 152 of Fig. 1.

8 9 Code Integrity Verification

10 Another tool for embedding some protection into a digital good is known as
11 “code integrity verification”. This tool defines one or more segments of the digital
12 good with “begin” and “end” labels. Each pair of labels is assigned an
13 identification tag. The tool computes a checksum of the data bytes located
14 between the begin and end labels and then hides the checksum somewhere in the
15 digital good.

16 Fig. 7 shows a portion of a digital good 700 having two segments S1 and
17 S2. In the illustration, the two segments partially overlap, although other
18 segments encoded using this tool may not overlap at all. The first segment S1 is
19 identified by begin and end labels assigned with an identification tag ID1, or
20 Begin(ID1) and End(ID1). The second segment S2 is identified by begin and end
21 labels assigned with an identification tag ID2, or Begin(ID2) and End(ID2).

22 The code integrity verification tool computes a checksum of the data bytes
23 between respective pairs of begin/end labels and stores the checksum in the digital
24 good. In this example, the checksums CS1 and CS2 are stored in locations that are
25 separate from the checkpoints.

1 The tool inserts a checkpoint somewhere in the digital good, outside of the
2 segment(s). Fig. 7 illustrates two checkpoints CP1 and CP2 for the associated
3 segments S1 and S2, respectively. Each checkpoint contains a function call to a
4 verification function that, when executed, computes a checksum of the
5 corresponding segment and compares that result with the precomputed checksum
6 hidden in the digital good. The checkpoints therefore have knowledge of where
7 the precomputed checksums are located. In practice, the precomputed checksums
8 CS1 and CS2 may be located at the checkpoints, or separately from the
9 checkpoints as illustrated.

10 When the client executes the digital good, the client-side evaluator 152
11 comes across the checkpoint and calls the verification function. If the checksums
12 match, the digital good is assumed to be authentic; otherwise, the client is alerted
13 that the digital good is not authentic and may be an illicit copy.

14 Acyclic (Dag-Based) Code Integrity Verification

15
16 Acyclic, or dag-based, code integrity verification is a tool that is rooted in
17 the code integrity verification, but accommodates more complex nesting among
18 the variously protected segments. “Dag” stands for “directed acyclic graph”.
19 Generally speaking, acyclic code integrity verification imposes an order to which
20 the various checkpoints and checksum computations are performed to
21 accommodate the complex nesting of protected segments.

22 Fig. 8 shows a portion of a digital good 800 having one segment S4
23 completely contained within another segment S3. The checkpoint CP4 for
24 segment S4 is also contained within segment S3. In this nesting arrangement,
25 executing checkpoint CP4 affects the bytes within the segment S3, which in turn

1 affects an eventual checksum operation performed by checkpoint CP3.
2 Accordingly, evaluation of segment S3 is dependent on a previous evaluation of
3 segment S4.

4 The acyclic code integrity verification tool 136(2) attempts to arrange the
5 numerous evaluations in an order that handles all of the dependencies. The tool
6 employs a topological sort to place the checkpoints in a linear order to ensure that
7 dependencies are handled in an orderly fashion.

8 9 Cyclic Code Integrity Verification

10 Cyclic code-integrity verification extends dag-based verification by
11 allowing cycles in the cross-verification graph. For example, if code segment S4
12 verifies code segment S5, and S5 also verifies S4, we have a cycle consisting of
13 the nodes S4 and S5. With such cycles, a proper order for checksum computation
14 does not exist. Thus, a topological sort does not suffice, and some checksums may
15 be computed incorrectly. Cycles require an additional step to fix up any affected
16 checksums.

17 One specific method of correcting checksums is to set aside and use some
18 "free" space inside protected segments. This space, typically one or a few
19 machine words, is part of the code bytes verified by checksum computation. If a
20 particular checksum is incorrect, the extra words can be changed until the
21 checksum becomes proper. While cryptographic hash functions are specifically
22 designed to make this impractical, we can use certain cryptographic message
23 authentication codes (MACs) as checksums to achieve this easily.

Secret Key Scattering

Secret key scattering is a tool that may be used to offer some security to a digital good. Cryptographic keys are often used by cryptography functions to code portions of a digital product. The tool scatters these cryptographic keys, in whole or in part, throughout the digital good in a manner that appears random and untraceable, but still allows the evaluator to recover the keys. For example, a scattered key might correspond to a short string used to compute indices into a pseudorandom array of bytes in the code section, to retrieve the bytes specified by the indices, and to combine these bytes into the actual key.

There are two types of secret key scattering methods: static and dynamic. Static key scattering methods place predefined keys throughout the digital good and associate those keys in some manner. One static key scattering technique is to link the scattered keys or secret data as a linked list, so that each key references a next key and a previous or beginning key. Another static key scattering technique is subset sum, where the secret key is converted into an encrypted secret data and a subset sum set containing a random sequence of bytes. Each byte in the secret data is referenced in the subset sum set. These static key scattering techniques are well known in the art.

Dynamic key scattering methods break the secret keys into multiple parts and then scatter those parts throughout the digital good. In this manner, the entire key is never computed or stored in full anywhere on the digital good. For instance, suppose that the digital good is encrypted using the well-known RSA public key scheme. RSA (an acronym for the founders of the algorithm) utilizes a pair of keys, including a public key e and a private key d . To encrypt and decrypt a message m , the RSA algorithm requires:

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

1 the decrypted block in memory. Simultaneously, a second process thread executes
2 and then destroys the code in the current block.

3 The benefit of this tool is that only one block is visible at a time, while the
4 other blocks remain encrypted. On the Intel x86 platform, code run in this manner
5 should be self-relocatable, which means that function calls are typically replaced
6 with calls via function pointers, or an additional program step fixes up any
7 function calls that use relative addressing. Other platforms may have other
8 requirements.

9 10 Code As An S-Box

11 Many ciphers, including the Data Encryption Standard (DES), use several
12 substitution boxes (S-boxes) to scramble data. An S-box is essentially a table that
13 maps n -bit binary strings onto a set of m -bit binary strings, where m and n are
14 small integers. Depending on the cipher, S-boxes may be fixed or variable. Both
15 S-boxes and code segments can be viewed simply as arrays of bytes, so an
16 important code segment can be used as an S-box for a cipher to encrypt another
17 important segment. If a cracker patches the segment serving as the S-box, the
18 encrypted segment will be incorrectly decrypted. This is similar in spirit to using
19 a segment's checksum as the decryption key for another segment, but is subtler
20 and better obfuscated.

21 22 Encryption/Decryption

23 Another tool to protect a digital good is encryption and decryption. This
24 tool breaks the digital good into different chunks and then encrypts each chunk
25

1 using different keys. The chunks might represent multi-layered and overlapping
2 code sections. Checksums of code sections can serve as encryption keys.

3 4 Probabilistic Checking

5 The secure processor has its own pseudorandom-number generator (PRNG)
6 that can be used to perform security actions, such as integrity verification, with
7 certain probabilities. Probabilistic checking uses these probabilities to ensure that
8 a protected program behaves differently during each run. For example, some
9 checks could be during every run, others approximately every other run, and still
10 others only occasionally. This makes the cracker's task much more difficult, since
11 a program no longer exhibits definite, repeatable behavior between runs. In fact, a
12 patched program may work properly once or twice, leading the cracker to believe
13 that his efforts were successful; however, the program will fail in a subsequent
14 run. This is part of an overall strategy of varying paths of execution between runs
15 to complicate reverse engineering, as described elsewhere in this document. .

16 17 Boolean Check Obfuscation

18 Boolean checking utilizes Boolean functions to evaluate the authenticity of
19 code sections or results generated from executing the code. A problem with
20 Boolean checking is that an attacker can often identify the Boolean function and
21 rewrite the code to avoid the Boolean check. According, the Boolean check
22 obfuscation tool attempts to hide the Boolean function so that it is difficult to
23 detect and even more difficult to remove.

1 Consider, for example, the following Boolean check that compares a
2 register with a value "1" as a way to determine whether the digital good is
3 authentic or a copy.

```
4 COMP reg1, 1  
5 BEQ good_guy  
6 (crash)  
7  
8 good_guy (go on)
```

9 In this example, if the compare operation is true (i.e., the Boolean check is
10 valid), the program is to branch to "good_guy" and continue. If the compare is
11 false, the program runs instructions that halt operation. To defeat this Boolean
12 check, an attacker merely has to change the "branch equal" or "BEQ" operation to
13 a "branch always" condition, thereby always directing program flow around the
14 "crash" instructions.

15 There are many ways to obfuscate a Boolean check. One approach is to
16 add functions that manipulate the register values being used in the check. For
17 instance, the following operations could be added to the above set of instructions:

```
18 SUB reg1, 1  
19 ADD sp, reg1  
20 :  
21 COMP reg1, 1
```

22 These instructions change the contents of register 1. If an attacker alters the
23 program, there is a likelihood that such changes will disrupt what values are used
24 to change the register contents, thereby causing the Boolean check to fail.
25

1 Another approach is to add “dummy” instructions to the code. Consider the
2 following:

3
4 LEA reg2, good_guy
5 SUB reg2, reg1
6 INC reg2
7 JMP reg2

8 The “subtract”, “increment”, and “jump” instructions following the “load
9 effective address” are dummy instructions that are essentially meaningless to the
10 operation of the code.

11 A third approach is to employ jump tables, as follows:

12 MOV reg2, JMP_TAB[reg1]
13 JMP reg2
14 JMP_TAB: <bad_guy jump>
15 <good_guy jump>

16 The above approaches are merely a few of the many different ways to
17 obfuscate Boolean checks. Others may also be used.

18 In-Lining

19 The in-lining tool is useful to guard against single points of attack. The
20 secure processor provides macros for inline integrity checks and pseudorandom
21 generators. These macros essentially duplicate code, adding minor variations,
22 which make it difficult to attack.
23
24
25

Reseeding of PRG With Time Varying Inputs

Many software products are designed to utilize random bit streams output by pseudo random number generators (PRGs). PRGs are seeded with a set of bits that are typically collected from multiple different sources, so that the seed itself approximates random behavior. One tool to make the software product more difficult to attack is to reseed the PRGs after every run with time varying inputs so that each pass has different PRG outputs.

Anti-Disassembly Methods

Disassembly is an attack methodology in which the attacker studies a print out of the software program and attempts to discover hidden protection schemes, such as code integrity verification, Boolean check obfuscation, and the like. Anti-disassembly methods try to thwart a disassembly attack by manipulating the code in such a manner that it appears correct and legitimate, but in reality includes information that does not form part of the executed code.

One exemplary anti-disassembly method is to employ almost plaintext encryption that indiscreetly adds bits to the code (e.g., changing occasional opcodes). The added bits are difficult to detect, thereby making disassembly look plausible. However, the added disinformation renders the printout not entirely correct, rendering the disassembly practices inaccurate.

Another disassembly technique is to add random bytes into code segments and bypass them with jumps. This serves to confuse conventional straight-line disassemblers.

Shadowing

Another protection tool shadows relocatable addresses by adding “secret” constants. This serves to deflect attention away from crucial code sections, such as verification and encryption functions, that refer to address ranges within the executing code. Addition of constants (within a certain range) to relocatable words ensures that the loader still properly fixes up these words if an executable happens not to load at its preferred address. This particular technique is specific to the Intel x86 platform, but variants are applicable to other platforms.

Varying Execution Path Between Runs

One protection tool that may be employed to help thwart attackers is to alter the path of execution through the software product for different runs. As an example, the code may include operations that change depending on the day of week or hour of the day. As the changes are made, the software product executes differently, even though it is performing essentially the same functions. Varying the execution path makes it difficult for an attacker to glean clues from repeatedly executing the product.

Anti-Debugging Methods

Anti-debugging methods are another tool that can be used to protect a digital good. Anti-debugging methods are very specific to particular implementations of the digital good, as well as the processor that the good is anticipated to run on.

As an example, the client-side secure processor may be configured to provide kernel-mode device drivers (e.g., a WDM driver for Windows NT and

2000, and a VxD for Windows 9x) that can redirect debugging-interrupt vectors and change the x86 processor's debug address registers. This redirection makes it difficult for attackers who use kernel debugging products, such as SoftICE. Additionally, the secure processor provides several system-specific methods of detecting Win32-API-based debuggers. Generic debugger-detection methods include integrity verification (to check for inserted breakpoints) and time analysis (to verify that execution takes an expected amount of time).

Separation in Time/Space of Tamper Detection and Response

Another tool that is effective for protecting digital goods is to separate the events of tamper detection and the eventual response. Separating detection and response makes it difficult for an attacker to discern what event or instruction set triggered the response.

These events may be separated in time, whereby tamper detection is detected at a first time and a response (e.g., halting execution of the product) is applied at some subsequent time. The events may also be separated in space, meaning that the detection and response are separated in the product itself.

Conclusion

Although the description above uses language that is specific to structural features and/or methodological acts, it is to be understood that the invention defined in the appended claims is not limited to the specific features or acts described. Rather, the specific features and acts are disclosed as exemplary forms of implementing the invention.